Memory Compression Techniques for Network Address Management in MPI

Yanfei Guo*, Charles J. Archer†, Michael Blocksome†, Scott Parker‡, Wesley Bland†, Ken Raffenetti* and Pavan Balaji*
*Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA
†Intel Corporation, Santa Clara, CA, USA
‡Argonne Leadership Computing Facility, Argonne National Laboratory, Lemont, IL, USA

Abstract—**MPI allows applications to treat processes as a logical collection of integer ranks for each MPI communicator, while internally translating these logical ranks into actual network addresses. In current MPI implementations the management and lookup of such network addresses uses memory sizes that are proportional to the number of processes in each communicator. In this paper, we propose a new mechanism, called AV-Rankmap, for managing such logical addressing. AV-Rankmap takes advantage of logical patterns in rank-address mapping that most applications naturally tend to have, and it exploits the fact that some parts of network address structures are naturally more performance critical than others. It uses this information to compress the memory used for network address management. We demonstrate that AV-Rankmap can achieve similar or better performance compared with that of other MPI implementations while using significantly less memory.**

I. Introduction

MPI is the most commonly used programming model for scientific computing on large supercomputing systems. Consequently, keeping up with the growing scale of such systems is a critical aspect in the design of MPI implementations. In the past few years, tremendous improvements have been made in MPI implementations with respect to avoiding data structures that scale linearly or superlinearly with system size [10]. Yet despite these improvements, MPI implementations today still have scalability limitations. One example is MPI's network address management.

MPI processes are logically represented as integer *ranks* within communicators, while the MPI implementation internally maintains the physical network addresses of the different processes. When a user application moves data between processes by addressing them using these ranks, the MPI implementation internally translates such ranks into the corresponding network addresses before communication can be performed. Such network address management has two closely related aspects that need to be considered. First, a ***network address*** for each peer process needs to be maintained: this includes the hardware communication address itself as well as other associated data structures. Second, a mapping between the logical communicator ranks to the network addresses, i.e., a ***rank-address mapping***, needs to be maintained for each communicator. Such structures exist in practically every MPI implementation today, even though the terminology used is sometimes different. For example, MPI implementations such as MPICH, MVAPICH, Intel MPI, Cray MPI, IBM Blue Gene MPI, Microsoft MPI, Tianhe MPI, and Sunway MPI, use the terminology of virtual connections (VCs) for network addresses and virtual connection reference tables (VCRTs) for the rank-address mapping structures. Open MPI and Fujitsu MPI, on the other hand, use the terminology of "proclist" and "plist", though conceptually they are no different from VCs and VCRTs.

Most MPI implementations manage network address and rank-address mapping structures in a way that is optimized for performance. For example, the number of dereferences or lookups are minimized, and few or no branches occur in the performance critical path. Unfortunately, not much attention has been paid to the memory usage of these data structures. For example, data structures related to different transports (e.g., network and shared memory) are embedded into the network address structures, rather than referenced from it. Similarly, data structures related to the shared-memory topology, which are required for transport selection, are embedded into the network address structures for fast lookup. Such a model, while optimized for performance, costs substantial memory.

This situation is also true for the rank-address mapping structures. Maintaining the rank-address mapping metadata is complicated by the fact that the MPI standard allows users to create new communicators by arbitrarily reordering the ranks compared to the parent communicator. That is, a single process can have two completely different ranks within two different communicators, with no correlation between the two. Because there are $P!$ (P factorial) valid mappings of communicator ranks to network addresses, where P is the number of processes in the communicator, lookup tables—such as VCRTs—remain the common practice for maintaining this metadata. Although this simple approach works for any possible reordering of ranks, it is not memory efficient. It takes $O(P)$ memory space on each process for each communicator, that is, $O(nP)$ total memory space on each process, where n is the number of communicators created by the process. Consequently, such metadata can result in the MPI implementation consuming a significant fraction of the available memory, particularly for very large supercomputers.

Our objective, in this paper, is to reduce the memory consumption of such network address management for MPI processes by using appropriate compression techniques. The important aspect here is not about how we can compress the network address management, but rather about how we can do so with virtually no performance degradation. For most MPI implementations performance is the most significant metric for measuring impact and generally any nonzero performance overhead is considered too high.

We propose a new mechanism, called AV-Rankmap, for network address management. AV-Rankmap uses several compression techniques to minimize the memory space used for network address management. We study the behavior of AV-Rankmap with respect to two properties: memory usage and performance.

With respect to memory usage, although AV-Rankmap does not improve memory usage in the worst-case scenario, it does significantly improve the common cases in which most applications fall. For example, we decouple the network address structure to distinguish elements based on various properties such as commonality of use, compressibility, and network-specific attributes. This decoupling allows applications that use only a subset of the features in MPI, without relying on the full generality of MPI, to benefit from a smaller overall memory footprint. Similarly, for applications that create communicators whose rank-address mapping follows certain patterns—three forms of which are studied in this paper: *direct*, *offset*, and *strided*—we detect such patterns and use them to reduce the rank-address mapping metadata storage. When we are unable to detect any pattern in the rank-address mapping, we simply fall back on the original lookup-table-based model.

With respect to performance, we study both the communicator creation path (which is typically not performance critical) and the data communication path (which is typically performance critical) and measure the overhead added in each case. For the noncritical path, we aim to keep the additional cost low, although some extra overhead is often tolerable. The additional cost is due primarily to the rank-mapping pattern detection involved in AV-Rankmap. We use simple pattern detection techniques, although one could use more sophisticated techniques that might have higher overhead.

For the performance-critical path, however, the overhead needs to be practically negligible for the proposed approach to be viable. For a more quantitative measure, a high-performance and finely-tuned MPI implementation would cost as little as $\sim$50 instructions on the communication path all the way from the application to the low-level network communication layer (e.g., in `MPI_Put`). Adding a single additional instruction in this path would lead to a 2% overhead in performance (assuming, for simplicity, that all instructions are equally expensive). Keeping this in mind, we perform a detailed instruction and cache-level analysis of the performance-critical path and illustrate that although AV-Rankmap adds a few additional instructions for address translation in the performance-critical path, the lost performance due to the additional instructions is more than compensated by the improved cache activity because of the smaller memory footprint, thus leading to similar or better performance while using significantly less memory.

The overall design of AV-Rankmap and a detailed evaluation with both microbenchmarks and real applications are showcased in this paper on up to 786,432 MPI processes. We demonstrate that AV-Rankmap can improve the memory usage of current MPI implementations by several orders of magnitude in many important cases.

II. Communicator Creation in MPI

At initialization time, MPI implementations create two communicators by default: `MPI_COMM_WORLD` and `MPI_COMM_SELF`. After initialization, applications are allowed to create additional communicators dynamically. A new (child) communicator is typically created from one existing (parent) communicator, although routines also exist that allow processes from two existing parent communicators to be combined into a single communicator.

Communicator creation routines can be broadly classified into four categories.

Duplicate Communicators. The most common communicator creation model used in MPI is duplication. MPI provides three routines in this model: `MPI_Comm_dup`, `MPI_Comm_idup`, and `MPI_Comm_dup_with_info`. These routines allow the application to create a new communicator with a rank-address mapping identical to that of the parent communicator but with a different communication context.

Split Communicators. The second communicator creation model is communicator splitting. Routines such as `MPI_Comm_split` fall into this category, in which the application can split the available processes into multiple disjoint groups (using a unique *color* to identify each group), where each group forms its own new communicator. In this model, the application can reorder the ranks of the processes in the new communicator, potentially in an arbitrary fashion. In practice, however, most applications do not reorder processes arbitrarily (or at all) when creating such split communicators. Thus, the resulting child communicators are often simple subsets of the parent communicator, with a well-structured mapping between the ranks of each process on the parent and child communicators. Other routines such as `MPI_Comm_create`, `MPI_Comm_create_group`, and `MPI_Comm_split_type` also fall in this category.

Topology-Aware Communicators. Topology-aware communicator creation routines such as `MPI_Cart_create`, `MPI_Graph_create`, `MPI_Dist_graph_create` and `MPI_Dist_graph_create_adjacent` allow applications to map application communication topologies to physical hardware topologies. For all these routines, two forms exist: with and without reordering. Reordering allows the ranks of the child communicator to be reordered in a topology-aware fashion (to allow for faster communication between some ranks compared with others). When reordering is disabled, the child communicator has a rank-address mapping identical to that of the parent communicator. In practice, however, topology-aware communicators are not widely used. In the cases where they are used, reordering is typically not applied. And in even when reordering is applied, most current MPI implementations do not perform any actual reordering of processes, with a few notable exceptions such as `MPI_Cart_create` on Blue Gene supercomputers.[1] The

[1]This is expected to change in the future: many MPI implementations are actively working on improving topology-aware communicator creation, and reordering ranks when needed.

outcome is that, in the vast majority of cases, the parent and child communicators have identical rank-address mappings. In cases where reordering is applied, the mapping is not identical, but is computable based on the hardware topology, although we do not do so in this paper.

Intercommunicators. Intercommunicators are created by using `MPI_Intercomm_create`, by dynamically spawning additional MPI processes (e.g., `MPI_Comm_spawn`), or by connecting multiple MPI applications (e.g., `MPI_Comm_connect`). An intercommunicator has two nonoverlapping groups: a remote group and a local group. Rank translations for these two groups are handled separately. Intercommunicators are rarely used in large applications. Most applications tend to rely on traditional communicators (called "intracommunicators") created by using a routine from one of the other three categories. The reason is threefold. First, typical application use cases do not map well to intercommunicators where communication is not as intuitive, e.g., a process can communicate with processes only in the remote group and not its local group. Second, not all routines in MPI are "intercommunicator-safe"; for example, one-sided communication windows are undefined over intercommunicators. Third, large supercomputers such as the IBM Blue Gene and Cray XE, XK, and XC systems do not support creating intercommunicators. As application workflows that allow multiple applications to connect to and communicate with each other gain more traction, we expect intercommunicators to be more widely used. Nevertheless, intercommunicators are unlikely to ever become the primary communication model for most applications.

An additional aspect to note here concerns internal communicators. In most MPI implementations, for every communicator that is created by the user (referred to as "user-visible communicators" in the rest of the paper), the MPI implementation internally creates additional convenience communicators. A common example is convenience communicators created for topology-aware collectives. For example, for each user-visible communicator, some MPI implementations create one additional internal subcommunicator containing processes that reside on the same node (i.e., a "node" communicator) and a second internal subcommunicator containing one root process from each node (i.e., a "node-roots" communicator). Thus, in essence, for each user-visible communicator a total of three communicators is created, although not all of them contain the same number of processes.

III. A Survey of Communicators in Applications

Section II described the different communicator creation techniques available in MPI. In practice, however, some techniques are more commonly used than others. To understand the communication creation models used in various applications, we performed a survey of a large number of applications comprising the NAS parallel benchmarks [5], CORAL benchmarks [3], DOE codesign applications [1], [2], and other large applications that consume significant compute cycles on large supercomputing centers [21], [6]. Although our survey covered 62 different applications, we highlight only a small subset of the survey here, because of space constraints.

Nek5000. Nek5000 is a highly scalable spectral-element method for solving computational fluid dynamics problems. Its computational model relies on solving computational grids at an increasing level of refinement. That is, for a given computational problem, it creates multiple grids, each at a different granularity or coarseness. Solving each grid gives an approximate solution to the problem. Solving the next-finer grid then refines the result based on the approximation generated by solving the previous coarser grid.

Because each grid requires its own communication context, Nek5000 creates a new communicator for each grid. These communicator are essentially duplicates of `MPI_COMM_WORLD`, and the number of communicators created increases as the number of levels of refinement desired by the application increases. A rule of thumb is that as the problem size grows, the number of refinement levels (and hence the number of communicators created) grows as well. In current production runs of the application, a "medium-scale" problem typically creates 24 refinement levels (i.e., 24 new communicators), and a "large-scale" problem typically creates 86 refinement levels (i.e., 86 new communicators). Medium-scale problems are considered appropriate to scale up to 16K–32K processes, while large-scale problems are considered appropriate to scale up to the full scale of the current largest supercomputers in the world.

NWChem. NWChem is a quantum chemistry application suite featuring a broad set of simulation capabilities targeted at many areas including quantum simulation of molecules with heavy isotopes and multiscale methods for modeling aqueous chemical reactions relevant to environmental chemistry. For its core linear algebra computations, NWChem (using libraries such as ScaLAPACK) creates virtual two-dimensional data grids on which the computation is carried out. To improve scalability, it splits the MPI processes into "row" and "column" communicators using `MPI_Comm_split`. Each process is a part of a "row" and a "column" communicator in this model. Data exchange and synchronization are then limited along these smaller communicators, thus improving performance and scalability. We note that during the split, process ranks are not reordered. Thus, the ranks of the processes in the "row" communicator are essentially the same as the ranks of the processes in `MPI_COMM_WORLD` but are offset by a constant value. Similarly, the ranks of the processes in the "column" communicator can be calculated with fixed offset and stride values. Such row/column communicators are common in other applications, such as QBOX [8], as well.

Apart from the core linear algebra computations, NWChem spends a large fraction of its computation on force calculations. These are typically done through one-sided communication that, after passing through multiple layers of the software stack, eventually uses MPI-3 one-sided communication (or RMA) windows internally. For each RMA window, the MPI implementation internally creates a new communicator that is a duplicate of the parent communicator from which the RMA window is being created, in order to perform the required data movement and synchronization. NWChem itself creates three to four RMA windows, depending on the

TABLE I: Mapping models for communicators.

Application	Dup	MPI_Comm_split			Topo	Intercomm
		Offset	Stride	Irreg.		
Nek5000	x					
QMCPACK		x	x			
NWChem	x	x	x			x
HACC					x	
QBOX	x	x				
CAM-SE	x	x				
NAMD	x	x				
LSMS		x		x		
SP	x	x				
BT	x	x	x			
FT	x	x				
Graph500	x	x	x			
Nekbone	x					
SNAP	x				x	
MCB	x					
cian2		x				
MCCK		x	x			
mocfe_bone		x	x			
pynamic	x	x	x			
MACSio	x					
AMG2013	x					
CNS	x					
SMC	x					
AMR	x					

problem being solved. However, when used together with the Casper [19] software stack for asynchronous progress, for each window created by NWChem, Casper creates as many duplicate windows as the number of cores on each node of the machine. For example, when NWChem is executed on the IBM Blue Gene/Q with 16 processes on each node, it creates 64 RMA windows (and thus 64 new communicators). When it is executed on the Intel Xeon Phi Knights Landing with 60–70 processes on each node, it creates on the order of 300 RMA windows (and thus 300 new communicators).

HACC. HACC is an astrophysics framework that simulates the formation of structure in the expanding universe. HACC's computational model is similar to the linear algebra portion of NWChem in that they both rely on multidimensional data grids for their computation. However, HACC does not split its communicators; instead, it creates multiple topology-aware Cartesian communicators (using `MPI_Cart_create`) representing three-dimensional, two-dimensional, and one-dimensional distributions of the problem. Like NWChem, HACC does not reorder the processes in the new communicator. Thus, apart from the additional topology information that is attached to the communicator, the process mapping itself is identical to that of the parent communicator.

Summary. Table I summarizes the communicator creation models used in a number of applications. In this table, we have classified the communicator creation models into several categories. The "dup" column refers to the cases where a duplicate communicator is created either directly by using one of the communicator duplication functions or indirectly by, for example, creating an RMA window on a communicator. "Offset" refers to the case where the process ranks in the new communicator are identical to that of the parent communicator, but at a fixed offset; "stride" refers to the case where the process ranks in the new communicator can be calculated based on a fixed offset and stride from the parent communicator; and "irreg" refers to the case where no pattern in the mapping is detected. The "topo" column refers to the cases where a topology-aware communicator is created. The "intercomm" column refers to the case where intercommunicators are used.

Based on the summarized information in Table I, we note a few important points.

1. The most commonly used communicator creation model is that of communicator duplication (either by using `MPI_Comm_dup` or by creating RMA windows). While the duplication itself is straightforward, as noted in Section II, the MPI implementation creates multiple internal communicators for each user-visible communicator. These internal communicators, however, are not simple duplicates of the parent communicator. Thus, in some sense, no "pure" duplication of communicators is possible in modern MPI implementations.

2. Split communicators (particularly, offset based and stride based) are heavily used. The offset-based model is the more common model and is valuable in splitting a multidimensional process grid of any number of dimensions. A common scenario where the offset-model of splitting is used is when the application wants to divide the available processes into smaller groups of fixed sizes. The stride-based model, on the other hand, is valuable in splitting a multidimensional process grid when the number of dimensions is larger than one.

3. Irregular splitting of communicators occurred in a single application, LSMS. We note, however, that here "irregular" does not mean that the application does not have any pattern in splitting the communicator. In fact, a real application is unlikely to create a split communicator with no pattern whatsoever. Here "irregular" means only that the pattern used by the application does not fall into the fixed set of patterns that we automatically detect. LSMS uses a master-worker scheme to parallelize the calculation for the electronic structure of large systems. The application divides its available processes into a single master process and multiple *walkers*; each *walker* is a collection of a small number of worker processes that together perform the required core computations. During initialization, LSMS creates one subcommunicator for each walker by splitting `MPI_COMM_WORLD`. These subcommunicators themselves are regular with their ranks mapping to the network addresses at a simple offset. However, when the MPI implementation creates its internal *node* and *node-roots* communicators, such mapping becomes more complicated. Figure 1 shows an example of the irregular communicator created in this splitting process. In this example, the communicator for the first walker (walker 0) has 12 processes and has an *offset* mapping model. Because the processes of the walker 0 reside on four nodes, MPI needs to create an internal *node-roots* communicator with four ranks. Ranks 0, 1, 2, 3 of the *node-roots* communicator are mapped to processes 1, 4, 8, 12. This mapping pattern does not match any pattern that we currently detect. Therefore, the MPI library *assumes* that this pattern is "irregular" and creates a lookup table for it. We note that the mapping of ranks 1, 2, 3 actually matches the *stride* mapping model, so we could have detected this pattern as a *stride* model with an additional special rank 0, but we do not currently do so.

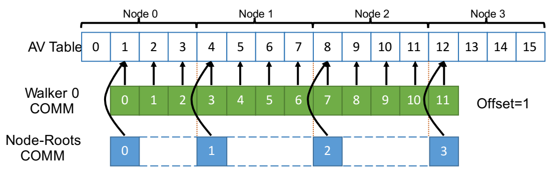


Fig. 1: An Example of LSMS Walker's Communicator and the "Irregular" Rank-Address Mapping in its *node-roots* Communicator.

4. Topology-aware communicators are used by two applications: HACC and SNAP [9]. Both use `MPI_Cart_create` to create these communicators, and neither application reorders the process ranks. Thus, the topology-aware communicators in these cases are essentially regular duplicate communicators, as far as process mapping is concerned.

5. Intercommunicators are rarely used and were observed only in NWChem. However, the use of intercommunicators in NWChem is not for creating dynamic processes; rather, it is a mechanism for creating regular intracommunicators, but in a noncollective fashion. MPI-2 did not provide functionality to create communicators in a noncollective fashion, and hence researchers attempted to do so by using intercommunicators, as described in [12]. In fact, the intercommunicators created in this process are temporary and are freed as soon as the final intracommunicator is created. MPI-3 included this functionality in `MPI_Comm_create_group`, which is simpler and more efficient compared with the method used by NWChem. At the time of our writing this paper, however, NWChem has not yet been updated to use this new MPI-3 functionality. When it does move to this new MPI-3 functionality, it will no longer have a dependency on intercommunicator creation. So, in a way, this dependency is a temporary artifact of the current implementation of NWChem.

IV. Design of AV-Rankmap

In this section, we first focus on the traditional VC-VCRT approach, the data structures it maintains and the advantages of such a model. Then we describe AV-Rankmap, including the overall design of the proposed approach, how it differs from VC-VCRT, and its benefits and disadvantages compared with VC-VCRT.

A. Traditional VC-VCRT Approach

The traditional VC-VCRT approach used in most MPI implementations uses a simple two-level hierarchy. At the top level is a VCRT structure, which essentially is a collection of pointers to each VC structure. The VCRT is an $O(P)$ structure that is statically allocated at initialization time.

At the bottom level is a VC structure that contains the required information for communicating with a process. The VCs themselves can be fully dynamically allocated, in theory, for example at the time of the first communication with the corresponding process. However, most MPI implementations today choose to statically allocate a small part of the VC (basic bookkeeping information) and dynamically allocate the more expensive portions of the VC on demand (such as network connections and communication buffers). Thus, the VCs result in another $O(P)$ memory space.

The VC is organized to minimize the number of dereferences. Consequently, all the information required for communication in embedded into this structure, rather than referenced from it. We classify the elements of the VC into three categories: core network access information, multitransport functionality, and functionality for dynamic processes.

Core Network Access Information. The core network access information refers to the basic network-specific functionality that is necessary for accessing a remote process. This includes information such as target endpoint information. For example, for InfiniBand, this would be the queue-pair information to which we can send data. Such information is the most basic and essential part of the VC and is required for any communication operation.

Multitransport Functionality. Almost every MPI implementation allows for data to be communicated over multiple transports. At least two transports are provided by all MPI implementations—shared memory for intranode communication and a network interconnect for internode communication—although some MPI implementations allow for more than two transports to be used simultaneously. Each transport has its own collection of information, such as communication functionality to use, communication thresholds for eager/rendezvous communication and queues for temporary communication buffers. For fast lookup, such information is directly embedded into the VC structure itself, thus avoiding a dereference. The cost of doing so, however, is that (1) the transport-specific information is replicated a large number of times across the different VCs and (2) some VCs might maintain more information than what they need for communication with the peer process that they correspond to (despite minimizing such additional information using unions).

Functionality for Dynamic Processes. Dynamically spawned or connected processes is a core part of the MPI standard, although it is rarely used in applications. However, current VCs tend to give such functionality importance (with respect to performance) equal to that of more commonly used functionality such as basic send/receive communication. Consequently, elements that are required to implement dynamic processes are embedded into the VC structure as well and use up memory space irrespective of whether the application uses dynamic processes or not.

B. AV-Rankmap: Address Vector Elements

AV-Rankmap retains the concept of network address and rank-address mapping structures, and instantiates them with data structures called *Address Vector Elements (AVEs)* and *Rankmap*. These would be logically equivalent to VCs and VCRTs, respectively, in the VC-VCRT model. AVEs reduce the size of the network address, compared with VCs, based on various properties such as commonality of use, compressibility, and network-specific attributes. The collection of AVEs for all peer processes is together referred to as the *Address Vector (AV)*. Rankmap reduces the memory footprint required for rank-address mapping, where possible, by detecting common rank-address mapping patterns, unlike VCRTs that always allocate an $O(P)$ lookup table.

In this section, we discuss AVEs and the AV. As described in Section IV-A, the traditional VC structure has three classes of components. Of these, the core network access information is the most critical part and is retained mostly as-is in the AVE structure. The only, relatively minor, difference is that AVE allows the actual network address to be either directly embedded into the structure (if it is small) or dereferenced from it (if it is large enough that it needs to be dynamically allocated on-demand).

Multitransport and dynamic process functionality, on the other hand, are significantly compressed, compared with VCs, as we demonstrate below.

Compressing the Multitransport Functionality. As mentioned in Section IV-A, traditional VCs maintain functionality associated with multitransport communication within the VC structure itself. This is highly redundant since the number of transports used is typically much smaller than the number of VCs. The number of VCs is equal to the total number of processes in the system, while the number of transports is equal to the number of networks being used (which is typically just two: shared memory and an internode network). Consequently, if we can decouple the transport-specific functionality, such information can be highly compressible. The challenge, however, is that such decoupling needs to be done in a way that it still retains fast lookup of this data, which was the primary reason these fields were embedded into the VC in the traditional model.

In AV-Rankmap, we carefully decouple such functionality from the corresponding AVE structure by considering two sets of transport-specific variables: (1) a single variable that identifies which transport to use and (2) a collection of variables that are used by the transport itself. These two sets have very different properties.

Identifying Which Transport to Use. The variable that identifies which transport must be used for a given peer process should either be embedded directly into the AVE (similar to what we do in VCs) or be easily and quickly computable. Fast computability is, unfortunately, not easy particularly when the processes are not laid out in a homogeneous manner. Thus, we chose to always store this information inside the AVE structure, using just enough bits to store the number of available transports (single bit for two transports). While, in theory, this is an $O(P)$ data structure, in practice, network transport addresses tend to have unused bits that can be used here without adding additional memory overhead. For example, the libfabric [7] network API allows network transports to use 63-bit network addressing, thus leaving behind one bit for such transport-selection functionality. Similarly, the UCX [18] network API uses aligned pointers for network addressing where the last 2-3 bits are unused (depending on whether the alignment is 4-byte or 8-byte). Extracting this information at runtime requires a bit-mask or bit-shift operation, which is a single (fast) instruction on most architectures.

Accessing Transport-specific Information. Decoupling transport-specific information from the AVE structure improves compressibility, but it will add an additional address dereference (e.g., a pointer lookup) to access this information. There is no escaping this dereference, unfortunately. But we can attempt to minimize its cost.

There are two costs associated with this additional dereference: (a) cache penalty for looking up the additional information, and (b) instruction costs (or instructions per cycle). Of these, based on our analysis, we expect the cache penalty to not be a significant issue. Specifically, for applications that perform frequent communication, these fields would already be in the processor cache anyway, and embedding them inside the AVE structure or not does not add any additional penalty as long as the processor cache has sufficient associativity. On the other hand, for applications that do not perform frequent communication and might not be able to retain the transport-specific information in their cache, the communication cost itself would likely not be as big a concern and the additional cache-miss to access this information would likely not be as important.

For the instruction costs, however, we have not yet been able to identify a satisfactory solution. The additional dereference results either in additional instructions (to load the transport-specific information to registers) or in more expensive instructions (e.g., memory-based instructions rather than register-based instructions). Even if the data is in cache, memory-based instructions seem to be fairly expensive compared to register-based instructions thus impacting the instructions-per-cycle that we can achieve. While this is certainly a concern, as we will demonstrate later, the smaller memory footprint of AV-Rankmap leads to fewer cache misses, which more than compensates for such additional instruction cost. Thus, from an overall performance perspective, such decoupling of transport-specific information is still a win.

Deprioritizing Dynamic Processes. As described in Section IV-A, the traditional VC structure gives equal importance to all fields, irrespective of how widely they are used in applications. In particular, dynamically spawned or connected processes need additional information such as which process group they belong to. Embedding this information into AVE can improve performance, but it also increases the size of the data structure for applications that do not use them. In AV-Rankmap, we decided to deprioritize dynamic processes such that applications that do not use dynamic processes use lesser memory. However, this deprioritization comes at a cost: applications that do use dynamic processes can, in some cases, use more memory than VC-VCRT.

Specifically, communicators that contain a combination of processes such that some of them are from one `MPI_COMM_WORLD` and some others from a different `MPI_COMM_WORLD` (i.e., dynamically spawned or connected processes) need to maintain two pieces of information: which process group does the remote process belong to and what is its rank within that process group. In VC-VCRT, both these pieces of information were stored inside the VC. Thus, it would increase the base memory usage but would not add extra memory usage for each new communicator created. In AV-Rankmap, however, we move this information out of AVE and into the communicator structure. Thus, the base memory usage would be small, but if the ranks of the new communicator are

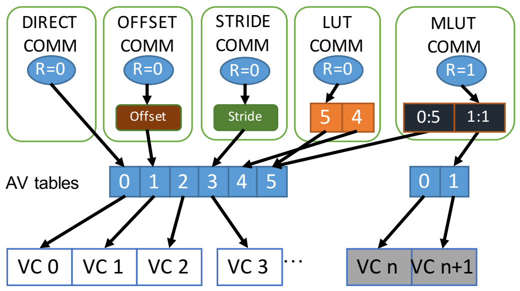


Fig. 2: Rank-Address Mapping.

arbitrarily reordered compared to the parent communicator, the incremental memory usage per communicator would be $O(P)$ where P is the size of the communicator. Fortunately, as we describe in Section IV-C, this is the worst-case scenario. In most cases, we can detect patterns in the formation of such communicators and can substantially compress this information.

Based on the above described compression techniques, we have been able to reduce the size of the AVE structure to 12 bytes compared with the 480 bytes needed by the VC structure: a 40-fold compression.

C. AV-Rankmap: Rank-Address Translation with Rankmap

Rank-address translation is essentially the process of finding the appropriate network address to communicate with given a communicator and a rank within that communicator. For any given process, the index of the AVE for that process is referred to as the *lpid* (local process ID). If the application spawns or connects to another MPI_COMM_WORLD, then a new AV is created that contains AVEs corresponding to the new group of processes. Each such spawned or connected group is referred to as a "process group" and has its own unique ID, called the *pgid*. A *pgid* refers to both the process group and the AV associated with that process group. Thus, a *pgid* and an *lpid* together can uniquely identify any process in the application. The rank-address translation process can be formally represented as the following mapping.

$< comm, rank > \rightarrow < pgid, lpid >$

Once the AV is created, the next step is to create a mapping between the communicator and one or more AVs. This is illustrated in Figure 2. As discussed in Section III, for the communicators created in most applications, the rank-address mappings are not arbitrary: they follow a simple, predefined pattern. AV-Rankmap takes advantage of this behavior to try to identify common patterns and use this information to compress the memory space required for maintaining such mapping. It identifies three regular mapping models: *direct*, *offset*, and *stride*, which represent the most common use cases in applications.

The ***direct*** model indicates that the ranks in the new communicator map to the same AV and its *lpids* in exactly the same order as MPI_COMM_WORLD. In this model, we do not need any additional storage other than the AV. The index in the AV (i.e., the *lpid*) is the same as the communicator rank in this model. Communicators that are duplicates of MPI_COMM_WORLD fall into this model. We note that in VC-VCRT such communicators still needs at least one $O(P)$ VCRT.

The ***offset*** model indicates that the ranks in the new communicator map to the same AV and its *lpids* in exactly the same order, but at a fixed offset, as MPI_COMM_WORLD. In this model, apart from the AV itself, the only additional piece of information that needs to be stored is the offset. The index in the AV (i.e., the *lpid*) can be calculated as the communicator rank plus the offset, in this model. Communicators that have been split without reordering from MPI_COMM_WORLD or one of its duplicates fall into this model.

The ***stride*** model allows the rank in a communicator to be mapped to a noncontiguous subgroup of MPI_COMM_WORLD with a fixed stride. The stride model has three parameters: stride, blocksize, and offset. The stride is the interval between the start of each block. The blocksize is the number of processes in each block. The offset is the offset of the rank 0 of the communicator.

All three regular mapping models use constant memory regardless of the number of ranks in the communicator. We note that these regular mapping models are valid only for communicators where all ranks are in the same process group, that is, there are no dynamic processes ($pgid = 0$).

When the rank-process mapping does not fit any of the regular mapping models, AV-Rankmap falls back on irregular mapping models using a rank lookup table. We designed two lookup tables for irregular mapping: *lut* and *mlut*. The lut is a dense array of lpids, similar to VCRT. It is used when all the ranks are in the same process group. The mlut is an array of $< pgid, lpid >$ pairs. It is used only when some of the ranks in the communicator belong to a different process group from others. We note that the lut requires only that all ranks have the same pgid; this pgid does not need to be zero. An example is an intercommunicator that is created when a new process group is connected. The remote group and the local group have different pgids, but the ranks in each group have the same pgid. Hence, both the remote group and the local group are represented by using lut instead of mlut. We need mlut only when we merge these two groups together using MPI_Intercomm_merge.

Note that there is an inclusive property in both the regular and the irregular mapping models. For example, a direct model can also be described as an offset model with the offset value equal to zero. This inclusive property is carefully exploited during communicator creation.

For each communicator, AV-Rankmap stores the mapping model and its corresponding metadata in a structure called *Rankmap*. As mentioned earlier, the Rankmap is logically equivalent to the VCRT, though the VCRT always uses a lookup table while the Rankmap can use more sophisticated techniques to achieve the same effect. By exploiting the regular patterns in the rank-address translation, the Rankmap only uses constant memory in the most common cases and $O(P)$ memory in the worst case. VCRT, by contrast, always uses $O(P)$ memory.

1) Creating the Rank-Address Translation for a Communicator: Users create new communicators based on existing communicators. When a new communicator is created, we

TABLE II: Child communicator mapping models for a given parent communicator mapping model (row) and *indirect mapping array* model (column).

	Direct	Offset	Stride	Irregular
direct	direct	offset	stride	lut
offset	offset	offset	stride	lut
stride	stride	lut	lut	lut
lut	lut[a]	lut[a]	lut	lut
mlut	mlut[a]	mlut[a]	mlut	mlut

[a] The child communicator does not create a new lookup table, but points to the lookup table of the parent communicator.

have two pieces of information: (1) the *indirect mapping array* that maps the ranks from the new communicator to the ranks in the parent communicator and (2) the compressed mapping model that allows us to translate a rank in the parent communicator to the corresponding AVs and lpids. Our task here is to create a similar compressed mapping model that allows us to translate a rank in the new communicator to the corresponding AVs and lpids. This is done in three steps. First, we try to detect a pattern in how the ranks in the child communicator correspond to the ranks in the parent communicators, i.e. the mapping model of the *indirect mapping array*. Second, we use this detected pattern together with any mapping pattern that exists between the parent communicator and the AVs/lpids, to generate a new mapping pattern between the child communicator and the AVs/lpids. Table II shows the possible child communicator's mapping model for each combination of the *indirect mapping array* model and parent communicator's mapping model. Third, if there are multiple parent communicators and the child has an irregular mapping, we try reduce it to one of the regular mapping models.

During the creation of a communicator, the MPI implementation first creates an *indirect mapping array* between the ranks in the child and the parent communicators. This information directly follows from the communicator creation function being used. Once this information is obtained, the next step is to convert this indirect mapping array to a compressed mapping pattern. To this end, we assume that the mapping pattern is offset based, and we calculate the offset value based on *rank 0* in the child communicator. We then try to validate the offset value with the remaining ranks in the indirect mapping. If successful, we set the mapping pattern between the child and parent communicator ranks to be offset-based. If not, we move on to stride mode and attempt to calculate the possible block size; and we follow a similar validation process before finally falling back to the irregular model if the stride mode cannot be validated.

After detecting the mapping pattern between the child and parent communicator ranks, the next step is to detect the mapping pattern between the child communicator ranks and the AVs/lpids. Recall that the mapping model of a communicator describes how the ranks are translated to AV table indices. Therefore, the mapping model of a child communicator is determined by both the rank-address mapping of the parent communicator and the pattern of the mapping between the child and parent communicators. Table II shows the state machine for determining the mapping pattern of the child communicator.

If the child communicator has an irregular mapping model (lut or mlut), the ranks are not necessarily irregularly ordered. The child communicator might have reordered the ranks in the parent communicator to create a regular mapping between the ranks and AV indices. AV-Rankmap performs an additional scan of the ranks to determine whether an irregular model can be converted to a regular mapping model. After all the indirect mappings are processed and the rank-address mapping is created, AV-Rankmap scans the lookup table using the same algorithm in step 1 above.

We note that communicator creation is not in the performance-critical path of most applications. Thus, while we do not want to make communicator creation excessively expensive, some additional cost to detect rank-mapping patterns and optimize them is often acceptable. As we will demonstrate in Section V-C, this overhead is between 3-7% in our implementation.

2) Accessing the Rank-Address Mapping: In this section, we discuss how the the rank-address mapping is accessed in AV-Rankmap. This translation is accessed inside the performance-critical path, and hence any overhead created in this path can slow down practically every performance-critical operation in the MPI implementation. So we need to be particularly cautious with respect to the performance overheads of our implementation choices in this path.

The translation is done in three steps: 1) checking the mapping model of the communicator; 2) calculating the corresponding index in the AV table for the rank; 3) accessing the AVE structure for the network address. In order to understand the overhead of our implementation, we use the Intel Software Development Emulator (SDE) [4] to obtain the instructions that are being executed for the translation. We studied three different implementations for such translation.

The first and most intuitive implementation of the rank-address translation is using a `switch` statement where each case contains the translation code for a specific model. Figure 3 and Figure 4 show the assembly code for the translation in VC-VCRT and AV-Rankmap. For a communicator with *direct* mapping, AV-Rankmap uses two additional instructions compared to VC-VCRT. For other mapping models, like *offset* and *stride*, there are additional instructions for calculating the index from the communicator rank. For example, Figure 5 shows the instructions for network address lookup in a *stride* communicator. While it has the same code for checking the mapping model and accessing the AVE, it has four more instructions (lines 6–9 in Figure 5) for loading the stride and offset values, and calculating the index of AVE. The use of the `switch` statement introduces four additional instructions (lines 1–4 in Figure 4). But we save two instructions for accessing the VCRT (lines 1–2 in Figure 3). The difference in the rest of the code is for the addressing changes in accessing the AVE. Note that the `jnbe` instruction in the switch statement is the branch to the default case. This is an unfortunate overhead because the AV-Rankmap approach does not have a "default" case and it is not possible to explicitly tell the compiler not to add this branch.

The second alternative implementation that we studied is to

```
mov  rax, qword ptr [rbp+0x60]
mov  rdx, qword ptr [rip+0x43566d]
mov  rax, qword ptr [rax+0x188]
mov  eax, dword ptr [rax+r12*4+0xc]
shr  eax, 0x1
cdqe
mov  rax, qword ptr [rdx+rax*8+0x8]
```

Fig. 3: Rank-Address Translation in VC-VCRT.

```
cmp    dword ptr [rdi+0x1a8], 0xa
jnbe   0x435039
mov    eax, dword ptr [rdi+0x1a8]
jmp    qword ptr [rax*8+0x5cfcc8]
movsxd rax, esi
add    rax, 0x1
shl    rax, 0x4
add    rax, qword ptr [rip+0x4682c6]
mov    rax, qword ptr [rax]
```

Fig. 4: Rank-Address Translation in AV-Rankmap (*direct* mode in the `switch`-based implementation).

```
cmp    dword ptr [rdi+0x1a8], 0xa
jnbe   0x435039
mov    eax, dword ptr [rdi+0x1a8]
jmp    qword ptr [rax*8+0x5cfcc8]
movsxd r12, esi
mov    eax, dword ptr [rdi+0x1b0]
imul   eax, r12d
add    eax, dword ptr [rdi+0x1b4]
cdqe
add    rax, 0x1
shl    rax, 0x4
add    rax, qword ptr [rip+0x4682c6]
mov    rax, qword ptr [rax]
```

Fig. 5: Rank-Address Translation in AV-Rankmap (*stride* mode in the `switch`-based implementation).

```
mov    edx, dword ptr [rdi+0x1a8]
cmp    edx, 0x1
jnz    0x435370
movsxd rax, esi
add    rax, 0x1
shl    rax, 0x4
add    rax, qword ptr [rip+0x467e8b]
mov    rax, qword ptr [rax]
```

Fig. 6: Rank-Address Translation in AV-Rankmap (*direct* mode in the hybrid implementation).

```
1  mov    edx, dword ptr [rdi+0x1a8]
2  cmp    edx, 0x1 ;if mode == direct
3  jnz    0x435370
4  cmp    edx, 0x3 ;if mode == offset
5  jnz    0x43537
6  cmp    edx, 0x5 ;if mode == stride
7  jnz    0x43538C
8  movsxd r12, esi
9  mov    eax, dword ptr [rdi+0x1b0]
10 imul   eax, r12d
11 add    eax, dword ptr [rdi+0x1b4]
12 cdqe
13 add    rax, 0x1
14 shl    rax, 0x4
15 add    rax, qword ptr [rip+0x467e8b]
16 mov    rax, qword ptr [rax]
```

Fig. 7: Rank-Address Translation in AV-Rankmap (*stride* mode in the hybrid implementation).

```
mov    eax, dword ptr [rdi+0x1a8]
jmp    qword ptr [rax*8+0x5cfbc8]
movsxd rax, esi
add    rax, 0x1
shl    rax, 0x4
add    rax, qword ptr [rip+0x47b1c0]
mov    rax, qword ptr [rax]
```

Fig. 8: Rank-Address Translation in AV-Rankmap (*direct* mode in the `goto`-based implementation).

use a hybrid `if` branch combined with a `switch` statement. That is, we can simply check if the communicator uses the *direct* mapping model (the `if` branch) and check for other mapping models using a `switch` statement in the `else` branch. This implementation can reduce the cost of translating the *direct* mapping model to eight instruction (as shown in Figure 6). However, the downside of this approach is that all other models in the `switch` statement will have two additional instructions due to the earlier `if` statement. We can further cascade multiple `if` statements to prioritize all the popular models and leave other mapping models in a `switch` statement. But for every level of the cascade, we would add two additional instructions, thus negating the benefit of a direct check after the first level. For example, the number of instructions for network address lookup on a *stride* communicator using the `if-switch` hybrid implementation is shown in Figure 7. Lines 2–7 show the instructions for three cascaded `if` statements. Along with the four instructions for calculating the AVE index, the network address lookup for a *stride* communicator would use eight more instructions.

In addition to these two models, we also studied a third implementation that, in essence, manually recreates the branch table that the compiler uses inside a `switch` construct, by using `goto` statements. As one might expect, our initial study (Figure 8) showed that this implementation has the least number of instructions for the branch lookup itself since it has the same two-instruction lookup for all models (similar to `switch`) and allows us to manually disable any extra branches that we do not need, such as the branch to the default case that the compiler adds for `switch` constructs. However, while integrating this implementation into the overall MPI implementation, we encountered a subtle and unexpected issue. Specifically, when the compiler translates the switch statement to `goto` branches, such translation is done *after* the relevant function or macro inlining. Thus, the compiler *sees* all of the inlined functions at the same time and assigns different labels to each `goto` branch. By doing this manually, however, since we do not have information on what functions the compiler might choose to inline, we lose the ability to assign different labels to the various `goto` branches. Consequently, assigning statically decided labels for such branches would, in practice, cause the compiler to disable inlining for such functionality. This causes much more significant overhead compared to the additional branches in the previous two implementations. The rank-address translation itself would reduce to five instructions, but the additional function call (because of no inlining) costs an additional 17 instructions.

V. Evaluation

In this section, we experimentally compare VC-VCRT and AV-Rankmap from two perspectives: memory usage and performance.

We used two different test platforms for our evaluation. The first platform is the Mira supercomputer at Argonne National Laboratory, which is a 49,152-node IBM BG/Q system. Each node on Mira has 16 cores and 16 GB memory, which allows running 768K processes at the full system scale. Most of our experiments were performed on Mira. However, the IBM BG/Q environment does not provide some capabilities, such as MPI dynamic processes and special tools like the Intel SDE, which is only available on Intel processors. For experiments that needed these capabilities, we used the Argonne LCRC "Blues" cluster. Each Blues node has two Intel Xeon E5-2670 processors (8 cores on each processor) and 64 GB memory. We ran experiments on Blues up to 256 nodes (4K processes).

The baseline implementation for our comparison was MPICH 3.2 which uses VC-VCRT. The libraries and applications in all experiments are compiled using GCC 4.7.2 with the `-O2` option and are statically linked.

A. Memory Usage for Regular Communicators

In this section, we focus on comparing the memory usage of VC-VCRT and AV-Rankmap for regular communicators.

1) Split Intracommunicators: In this experiment, we used a benchmark that splits the odd and even ranks of `MPI_COMM_WORLD` into two subcommunicators without reordering the ranks. Thus, the ranks in the split communicator have the *stride* mapping model in AV-Rankmap. This process is repeated a number of times to create a fixed number of split communicators.

Figure 9(a) shows the memory usage of 10 and 100 split communicators with up to 768K processes in the parent communicator. It is clear that AV-Rankmap uses significantly less memory than VC-VCRT. At the full scale on Mira, AV-Rankmap uses only 9 MB of memory for 100 split communicators. VC-VCRT, on the other hand, consumes more than 40% of system memory for 10 communicators and exceeds the total system memory for 100 communicators.

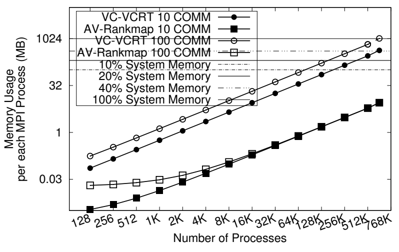
Figure 9(b) shows the breakdown of the memory usage for 10 communicators in AV-Rankmap and VC-VCRT with increasing number of processes in the parent communicator. We note that the memory usage of both approaches is $O(P)$ with respect to the total number of processes in the system: this is because both approaches need to store the network physical addresses, which takes $O(P)$ memory. But, AV-Rankmap has a memory usage advantage in two aspects. First, since the size of the network addresses used in AV-Rankmap is smaller (based on the implementation choices described in Section IV-B), the constant associated with the $O(P)$ increase in memory is smaller. In our implementation, we are able to reduce the size of each AVE to 12 bytes in AV-Rankmap, which is 40-times smaller than the 480 bytes used by each VC in VC-VCRT. Second, for the rank-address mapping: since AV-Rankmap does not use a lookup table in common communicator patterns, but instead dynamically computes the rank-address mapping (based on the implementation choices described in Section IV-C), in the common case we use constant memory instead of an $O(P)$ structure, while VC-VCRT uses an $O(P)$ structure for the lookup.

Figure 9(c) shows a different breakdown of the memory usage, this time keeping the number of processes fixed at 768K but increasing the number of communicators created. As expected, the memory usage of VC-VCRT grows quickly with the number of communicators because it uses a new $O(P)$ VCRT lookup table for rank-address mapping on each new communicator. In AV-Rankmap, on the other hand, each new communicator uses only a small constant memory space (e.g., to store the offset and stride, which are two integers) if its rank-address mapping is regular, as is the case in our benchmark.

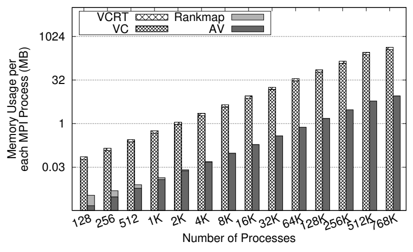
2) Duplicate Intracommunicators: In this experiment, we used a benchmark that simply duplicates `MPI_COMM_WORLD`. Thus, the duplicated communicators would have the *direct* mapping model in AV-Rankmap. As explained in Section II, for every communicator that is created by the user, the MPI implementation internally creates additional convenience communicators: a *node* communicator (for ranks on the same node) and a *node-roots* communicator (for root ranks in all nodes).

Figure 10(a) shows the overall memory usage for 10 and 100 duplicate communicators. Unlike the results for the split communicators, where the memory usage of the VC-VCRT approach is distinctly different for 10 and 100 communicators, for duplicate communicators the change in memory usage with increasing number of communicators is much smaller. This is because the baseline implementation (MPICH 3.2) already adopts an optimization technique for duplicate communicators [13]. That is, instead of copying the lookup table, each duplicate communicator only needs to maintain a pointer to the parent's lookup table. However, there is still an increase in memory usage. This is not because of the user-visible communicator itself, but because of the *node* and *node-roots* communicators that the MPI implementation internally creates for each user-visible communicator. These internal communicators are not simple duplicates of the parent communicator since they contain lesser processes and their rank layout is different from that of the parent communicator. This results in these internal communicators creating their own new lookup tables, and thus using additional memory.

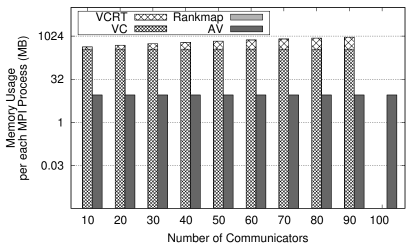
One possible solution that could have helped improve memory usage in this particular benchmark could be as follows: whenever a new communicator is created, search the existing lookup tables that were previously created to see if any of them could be reused for the new communicator. If any of them matched, the MPI implementation could simply reuse the previous lookup table instead of creating a new one. At least for simple duplication of communicators, this approach would have kept the memory usage fixed with increasing number of communicators—the internal *node* and *node-roots* communicators associated with the new user communicator would be able to reuse the VCRTs for the internal *node* and *node-roots* communicators associated with the parent communicator. While such an optimization would have optimized the case where many duplicate communicators are created, we consider such a solution to be too specific to duplicate communicators. AV-Rankmap, on the other hand, provides



(a) Different numbers of processes



(b) Detail memory usage of 10 communicators



(c) Detail memory usage with 768K processes

Fig. 9: Memory usage of `MPI_Comm_split` with different numbers of processes and communicators.

a more generic optimization for all regular communicators instead of the special case of duplicate communicators thus, in some sense, subsuming such optimizations. Moreover, as shown in Figures 10(b) and 10(c), the VCRT portion of the memory usage is fairly small compared with the VC portion—even at 100 communicators and 768K processes, the VC portion takes nearly 95% of the total memory usage. Thus, even with this optimization there would be very small overall gain in memory usage.

3) Regular Intercommunicators: In this experiment, we first split `MPI_COMM_WORLD` into odd and even communicators without reordering process ranks, and then create an intercommunicator between these split communicators. Finally, we either duplicate this intercommunicator or split it into odd/even ranks (again, without reordering process ranks) and measure the memory usage.

Figure 11 shows the memory usage of 10 and 100 split intercommunicators. We notice a similar trend for split intercommunicators as we did with split intracommunicators in Section V-A1. That is, AV-Rankmap uses significantly lower memory compared with VC-VCRT. One subtle difference between intra- and intercommunicators is that VC-VCRT uses more memory for intercommunicators, going up to ∼1.4 GB for 100 intercommunicators on 768K processes compared with ∼1 GB for 100 intracommunicators. This is because each intercommunicator needs to maintain two separate mappings—one for the local group and one for the remote group—thus requiring additional storage. Further, in addition to the *node* and *node-roots* communicators, each intercommunicator also has a *local* communicator for the ranks in the local group. These internal communicators have to maintain their own mappings as well, which further increases the difference in storage needed between VC-VCRT and AV-Rankmap.

Figure 12 shows the memory usage of 10 and 100 duplicate intercommunicators. As expected, the results of this experiment follow a similar trend as that with duplicating intracommunicators shown in Section V-A2. Again, the memory usage numbers for intercommunicators are slightly higher than that with intracommunicators. This is because of the same reason as described above for split intercommunicators.

B. Memory Usage for Irregular Communicators

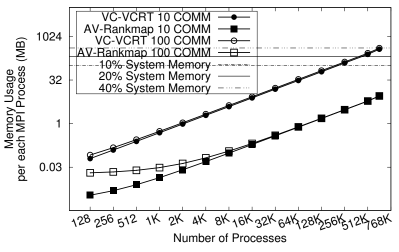
In AV-Rankmap, a communicator with an irregular rank-address mapping uses either the *lut* or the *mlut* model. Such irregular communicators occur either when the ranks of the communicator are reordered in a pattern that is not detected within the set of patterns that we look for, or when the application dynamically spawns or connects to additional MPI processes. Here we study the memory usage of these irregular communicators.

1) Communicators with Reordered Ranks: In this experiment, we create an irregular intracommunicator by duplicating `MPI_COMM_WORLD` and reordering the ranks in a random order. We ensure that each duplicated communicator has a unique rank order—this prevents any reuse of lookup tables between communicators. In this case, both AV-Rankmap and VC-VCRT have to allocate one lookup table for each communicator, i.e., $O(P)$ memory for the mapping structure. Figure 13 shows the memory usage of 10 and 100 such communicators. We note that both approaches consume a significant amount of memory due to their respective lookup tables. Moreover, the lookup tables are of the same size in both approaches, thus the rate of increase in memory usage with the number of communicators is the same in both approaches. However, the AVE itself is smaller in AV-Rankmap compared with the VC in VC-VCRT, which results in AV-Rankmap's base memory usage (which is independent of the number of communicators) to be lesser than that of VC-VCRT.

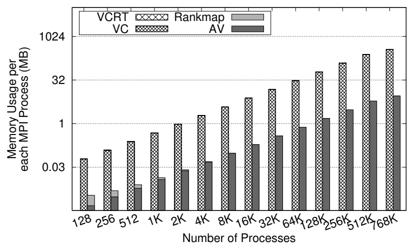
We remind the reader that this is the worst-case scenario for intracommunicators and has been showcased here only for completeness. In practice we do not see real applications creating communicators with such arbitrary reordering of process ranks (see Table I).

2) Communicators with Dynamic Processes: As discussed in Section IV-B, one of the techniques that we use to compress the AVE structure in AV-Rankmap is to deprioritize dynamic processes. That is, information specific to dynamic processes, i.e., process group ID mapping for each process, is moved out of AVE and into the communicator structure. The advantage of this approach is that for applications that do not use dynamic processes there is no additional storage requirement. The flip-side of this optimization, however, is that applications that do use dynamic processes are penalized more heavily since each new communicator that is not a simple remapping of the parent communicator ranks would now need to maintain the expensive *mlut* lookup table.

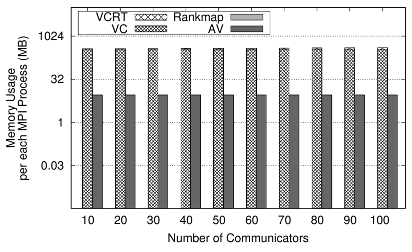
Here we present our evaluation for the case where a single dynamic-process communicator is created. We create an intercommunicator with dynamic processes and merge it into



(a) Different numbers of processes



(b) Detail memory usage of 10 communicators



(c) Detail memory usage of 768K processes

Fig. 10: Memory usage of `MPI_Comm_dup` with different numbers of processes and communicators.

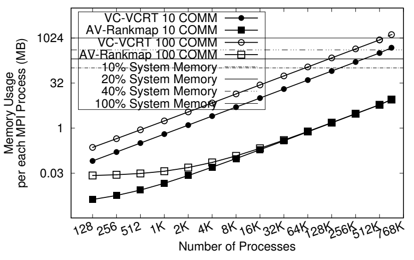


Fig. 11: Memory Usage of 10 and 100 Split Intercommunicators.

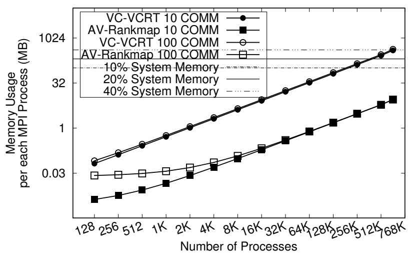


Fig. 12: Memory Usage of 10 and 100 Duplicate Intercommunicators.

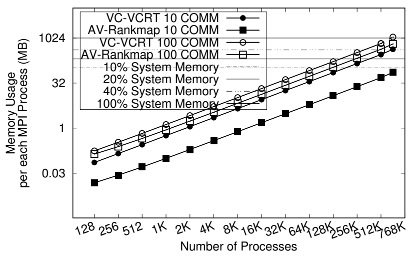


Fig. 13: Memory Usage of 10 and 100 Irregular Communicators.

an intracommunicator. The intracommunicator would thus use the *mlut* mapping model in AV-Rankmap. We then duplicate this intracommunicator and randomly reorder the ranks in the new communicator. The rank order in each new communicator is unique to prevent lookup table sharing. We used the Blues system for this experiment because the Mira supercomputer does not support spawning dynamic processes.

Figure 14(a) shows the memory usage for 10 and 100 such communicators in VC-VCRT and AV-Rankmap. While AV-Rankmap uses less memory than VC-VCRT for 10 communicators, it uses significantly more memory than VC-VCRT for 100 communicators. As Figure 14(b) shows, AV-Rankmap uses about twice the amount of memory to maintain the associated lookup table. This is because each element in the lookup table in AV-Rankmap stores both the AV table ID and the AV table index (*mlut* model), while the lookup table in VC-VCRT stores only the VC index. The memory usage trend is more clear when we fix the number of processes to 4096 and change the number of communicators. As shown in Figure 14(c), the memory usage of AV-Rankmap grows rapidly as we increase the number of communicators.

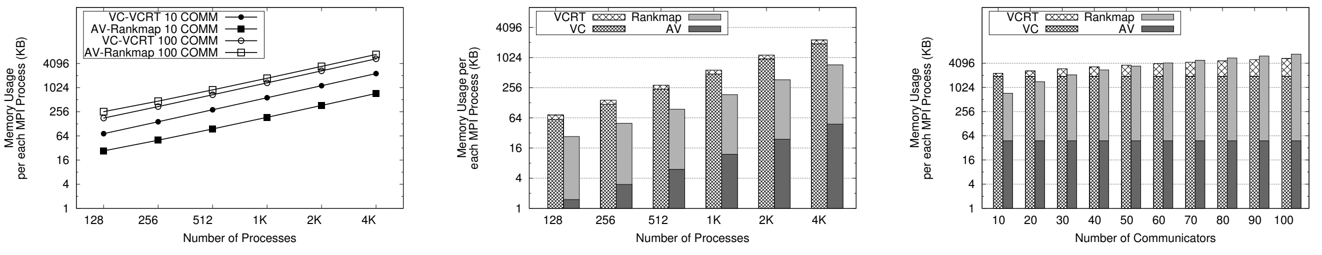
While this memory overhead is certainly an issue with the design choice made in AV-Rankmap, we believe this tradeoff is worthwhile for most applications since dynamic processes are rarely used in large MPI applications as discussed in Section III. We further note that for other communicators derived from an *mlut*-based communicator the previously discussed mapping compression techniques still hold. For example, simple duplicate communicators created from such a communicator would not create additional *mlut* tables and would use only constant memory space.

C. Performance

We study two performance aspects in this section: (1) communicator creation overhead and (2) network address lookup overhead. As discussed earlier, communicator creation is not on the performance-critical path for most applications. Thus, while we do not want to make it too expensive, some overhead is typically acceptable. Network address lookup, on the other hand, must show no observable overhead for the approach to be practically viable.

1) Communicator Creation Cost: In AV-Rankmap, when a new communicator is created, the MPI implementation checks to see if the rank-address mapping of the new communicator matches any of the regular patterns that it understands. If it does match a regular pattern, it stores the associated simple metadata (e.g., offset and stride) instead of creating a full lookup table for the network address translation. This check for regular patterns, however, adds some overhead to the communicator-creation path. In this section, we measure the communicator creation overhead for splitting communicators, duplicating communicators, creating/merging intercommunicators without dynamic processes, and creating/merging intercommunicators with dynamic processes. As mentioned earlier, Mira does not support dynamic processes. So experiments on Mira did not include intercommunicators with dynamic processes.

Figures 15 and 16 show the overhead on communicator creation time imposed by AV-Rankmap compared with VC-VCRT on Mira and Blues, respectively. Overall, AV-Rankmap adds 3%–6% overhead in communicator creation time on Mira and 3%–8% overhead on Blues. There are a few trends that we observe here. First, communicator creation time increases with the number of processes. This is expected since the checks to see whether the rank-address mapping is regular or not are linear with respect to the number of processes.



(a) Different numbers of processes (b) Detail memory usage of 10 irregular communicators (c) Detail memory usage of 768K processes

Fig. 14: Memory usage of mlut communicators with different numbers of processes.

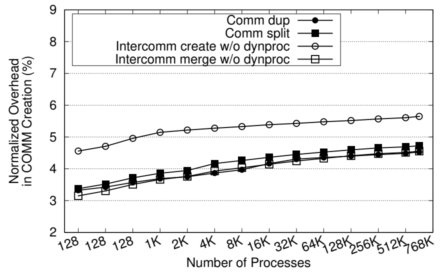


Fig. 15: Creation time for different communicators on Mira.

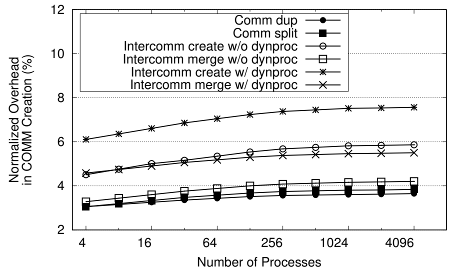


Fig. 16: Creation time for different communicators on Blues.

Second, creating intracommunicators with `MPI_Comm_dup` or `MPI_Comm_split` is a little faster than creating intercommunicators. This is because, for intercommunicators, AV-Rankmap needs to search for regularity in both the process group IDs (e.g., if they are all the same) as well as the process ranks. The overhead on Mira is slightly lesser than that on Blues because when running on Mira the exchange of network addresses between the local and remote groups during the intercommunicator creation is bypassed—we already know that on BG/Q platforms all the processes are from the same `MPI_COMM_WORLD`. This saves time for identifying new process groups, creating a new AV table and inserting the network addresses.

2) Network Address Lookup Performance: As mentioned in Section IV-C2, in order to lookup the network address corresponding to a communicator rank, the MPI implementation must first check what mapping model that communicator is using and then use that information to either compute or lookup the actual network address. Here we study the performance impact of AV-Rankmap on the network address lookup.

For this experiment, we developed a microbenchmark that issues one million `MPI_Put` operations, each with a message size of 8 bytes (one double), to each rank in the communicator in a round-robin fashion. The microbenchmark tests communicators in five mapping models: *direct*, *offset*, *stride*, *lut* and *mlut*. All five communicators have the same number of ranks (half of the ranks in `MPI_COMM_WORLD`) to ensure that all experiments send the same total number of messages. The *direct* communicator splits `MPI_COMM_WORLD` into two contiguous halves and performs the experiment on the first half communicator. The *offset* communicator also splits `MPI_COMM_WORLD` into two contiguous halves, but performs the experiment on the second half communicator. The *stride* communicator splits `MPI_COMM_WORLD` into odd/even ranks and performs the experiment on the odd half. The *lut* communicator is similar to the *stride* communicator, but with the ranks reversed. The *mlut* communicator is created by merging ranks from a dynamically spawned group of processes. We study the performance from three aspects: instruction-count, cache misses and message issue rate.

Instruction-count Analysis: For the instruction-count analysis, we added trace points (i.e., markers) before and after the network address lookup and used Intel SDE to record the instructions executed between the two markers. We measured the instruction-counts of three different implementations of rank-address mapping (as introduced in Section IV-C2). As noted earlier, Intel SDE is only available on x86 architecture, so these experiments used Blues.

Table III shows the instruction counts for rank-address mapping using the `switch`-based implementation, the `if-switch` hybrid implementation and the `goto`-based implementation. The baseline implementation is VC-VCRT (MPICH 3.2) which uses seven instructions. Much of the analysis of the different implementations of rank-address mapping is already presented in Section IV-C2, so we will not duplicate that analysis here. Instead, we will summarize our findings and recommendations here.

First, compared with the switch-based implementation, the hybrid implementation can reduce the instruction count needed for the *direct* mode by one instruction, but this comes at the cost of increasing the instruction-counts for all other types of communicators because of the additional `if` branches. Thus, only the *direct* communicator can benefit from such an implementation. However, as mentioned earlier, optimizing

TABLE III: Instruction Counts for Rank-Address Mapping.

	direct	offset	stride	lut	mlut
VC-VCRT	7	7	7	7	7
AV-Rankmap (switch)	9	11	13	11	15
AV-Rankmap (hybrid)	8	12	16	18	21
AV-Rankmap (goto)	22	24	26	24	28

for *direct* communicators alone is impractical: even when the application itself uses only *direct* communicators, MPI implementations internally create additional communicators which are not *direct* communicators. For this reason, we do not consider the hybrid implementation as a good choice for AV-Rankmap.

Second, while the instruction-count of the `goto`-based implementation for the rank-address translation itself is lower than the other implementations, as mentioned in Section IV-C2, the compiler cannot inline the rank-address mapping function in this case. Thus, while the rank-address mapping itself only costs five instructions, the additional function call costs 17 additional instructions. As a result, the instruction-count of the `goto`-based implementation is much higher than the other two implementations. Therefore, we do not consider the `goto`-based implementation as a good choice for AV-Rankmap either. In the following experiments, we only evaluate AV-Rankmap using the `switch`-based implementation.

Cache Analysis: As discussed in Section IV-B, AV-Rankmap uses multiple techniques to compress the network address structure. As a result, the AVE in AV-Rankmap is significantly smaller than the VC in VC-VCRT (12 bytes vs 480 bytes). While AV-Rankmap costs additional instructions as discussed above, its smaller memory footprint significantly reduces the cache misses incurred during communication. In this section, we study this cache impact. We performed two sets of experiments. The first experiment used just two processes (one process per node) within each communicator, while the second experiment used 16 processes (eight processes per node) within each communicator. In our experiments, each process issues 8 million `MPI_Put` operations to each of the other ranks in the communicator in a round-robin fashion. We measure the cache misses using PAPI on Blues and BGPM on Mira. The cache was warmed up using additional iterations of the same operation before our measurements.

Table IV shows the cache misses on Blues and Mira. The results show that VC-VCRT (baseline) experiences a significantly larger number of misses in L1D and L2 caches, as compared to AV-Rankmap. This is because of the smaller memory footprint of AV-Rankmap. The cache lines on both Blues and Mira are 64 bytes. Thus, each VC object in VC-VCRT is spread across eight cache lines. One AVE object in AV-Rankmap, on the other hand, is only 12 bytes and thus one cache line can fit multiple AVE objects.

With two processes, VC-VCRT experiences more than $\sim$40-fold higher number of L1D cache misses and 2–4-fold higher number of L2 cache misses compared with AV-Rankmap. This is a significant difference that is attributed to the smaller memory footprint of AV-Rankmap. With eight processes, the difference in cache misses goes up to nearly three orders of magnitude for L1D and $\sim$5-fold for L2. With a larger number of processes, because the microbenchmark issues `MPI_Put` to multiple ranks in a round-robin fashion, the cache impact is more pronounced.

TABLE IV: Per-process cache misses.

Cache	Baseline	direct	offset	stride	lut	mlut
L1D (8 Procs, Blues)	120491	103	168	141	98	192
L2 (8 Procs, Blues)	237	47	43	69	53	60
L3 (8 Procs, Blues)	47	48	47	46	47	45
L1D (8 Procs, Mira)	180491	184	191	189	178	N/A
L2 (8 Procs, Mira)	422	57	53	61	68	N/A
L1D (2 Procs, Blues)	4438	102	157	133	82	168
L2 (2 Procs, Blues)	107	47	45	64	56	58
L3 (2 Procs, Blues)	48	46	43	45	46	47
L1D (2 Procs, Mira)	6540	186	189	190	180	N/A
L2 (2 Procs, Mira)	218	56	54	61	66	N/A

Message issue rate: Here, we measure the message issue rate of VC-VCRT and AV-Rankmap for `MPI_Put`. We perform two experiments. The first experiment is on real networks, using Blues (InfiniBand network) and Mira (BG/Q network), to show the performance impact of AV-Rankmap in practice. The second experiment is a worst-case measurement for a theoretical infinitely-fast network; to emulate such a network we go through the entire MPI stack but bypass the actual network communication. This experiment is designed to help us understand how AV-Rankmap would behave on future, more-efficient, networks. Table V illustrates the message issue rate for both experiments where the top two lines correspond to the first experiment and the bottom two lines correspond to the second experiment. We used two nodes and performed experiments with two processes in each communicator (one process per node) and 16 processes in each communicator (eight processes per node). All `MPI_Put` messages are sent to ranks on the remote node.

In the first experiment (real network), the message issue rate is bound by the network speed on the machine and we note that AV-Rankmap and VC-VCRT show identical issue rates for all communicator types. So we can conclude that AV-Rankmap can significantly improve memory usage without hurting communication performance.

In the second experiment (theoretical infinitely-fast network), the message issue rate is only bound by the CPU processing speed and the processing cost of MPI. In this case, AV-Rankmap outperforms VC-VCRT in all scenarios achieving up to 50% higher maximum issue rate. The performance improvement is mainly because the improved cache locality of AV-Rankmap. The difference in maximum issue rate between different mapping models reflects the cost of the extra work in the rank-address mapping. The measured maximum issue rates concur with the results on the instruction-counts. That is, the *direct* model is expected to have the highest issue rate because it has the least overhead in the rank-address mapping. The maximum issue rate decreases as the instruction count increases from the *offset* model to the *mlut* model. Note that, the performance difference between Mira and Blues is mainly because Mira's CPU has a lower frequency (1.6 GHz vs 2.7 GHz of Blues) and fewer integer units (1 per core vs 4 per core on Blues).

TABLE V: Per process issue rate (million/second) with *switch*-based rank-address translation.

	Baseline	direct	offset	stride	lut	mlut
Mira (8 procs, real)	0.8999	0.8989	0.8994	0.8963	0.8969	N/A
Blues (8 procs, real)	4.325	4.338	4.334	4.322	4.324	4.318
Mira (8 procs, theoretical)	11.08	16.63	15.48	15.08	16.00	N/A
Blues (8 procs, theoretical)	64.76	98.79	92.13	89.64	96.03	84.51
Mira (2 procs, real)	0.8899	0.8879	0.8884	0.8863	0.8969	N/A
Blues (2 procs, real)	4.302	4.311	4.313	4.303	4.304	4.297
Mira (2 procs, theoretical)	13.08	16.63	15.48	15.08	16.00	N/A
Blues (2 procs, theoretical)	78.20	98.89	93.43	89.91	96.96	85.27

D. Applications

In this section we evaluate AV-Rankmap with the applications presented in Section III. Experiments were performed with all applications presented in the previous sections, but many of them follow similar trends so we did not find it necessary to show results for all of them in this paper. Instead, we show the results for the *Nek5000* application, and several miniapps including *BT*, *FT*, and *SP*, from the NAS parallel benchmarks [5]; *AMG2013* and *Nekbone*, from the CORAL benchmarks [3]; and *AMR*, from the ExACT codesign center [2].

We conducted experiments to measure both the performance and memory usage of the various applications with VC-VCRT and AV-Rankmap. There was no observable difference in performance between the two approaches similar to what we noticed in Section V-C2. Hence the performance results for the applications are not shown in this paper. Instead, we will focus on the memory usage results. All applications were run on 512K processes of Mira.

We first study the impact of AV-Rankmap on *Nek5000*. We evaluate Nek5000 with two problem sizes: medium and large. The medium-sized problem uses the XXT solver, which creates 24 duplicates of `MPI_COMM_WORLD`. The large-sized problem uses the AMG solver, which creates 86 duplicates of `MPI_COMM_WORLD`. Figure 17 presents the per-process memory consumption for the network address management functionality. On 512K processes, VC-VCRT uses around 250 MB per process, which is nearly 25% of the memory available on the node. With AV-Rankmap, on the other hand, the memory usage is negligible. The figure also shows the breakdown of the memory used by the VCs and VCRTs. The memory size for VC portion of the memory usage is identical for both problem sizes because it depends on the number of processes, which is the same for both problem sizes, and not the number of communicators. The memory size of the VCRT portion of the memory usage, on the other hand, depends on the number of communicators created which is higher for the larger problem.

We next study the memory usage of the miniapps discussed above, in Figure 18. Overall, most of the miniapps that we studied create between two and seven user communicators which is a small number. So the amount of memory used by the VCRTs, which depends on the number of communicators, is small. The amount of memory used by the VCs, on the other hand, depends on the number of processes and is the dominating factor in the overall memory usage. The memory usage of these miniapps on 512K processes is 244 MB (∼25% of the system memory) with VC-VCRT. In contrast, AV-Rankmap uses only 4 MB memory on 512K processes.

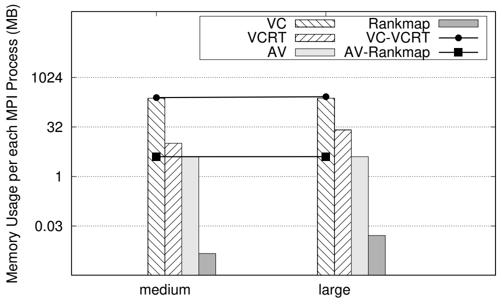


Fig. 17: Network address management memory consumption for Nek5000 (512K processes).

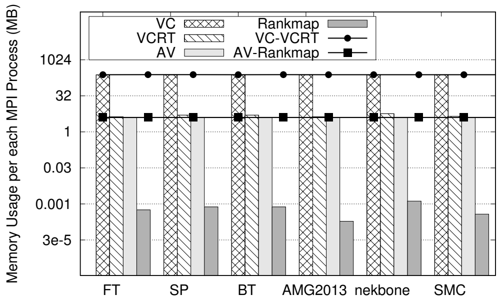


Fig. 18: Memory usage of miniapps on 512K processes.

While the memory usage of the miniapps is dominated by that of the VCs, we would like to draw the reader's attention to the memory used by VCRTs for the *Nekbone* miniapp. Specifically, *Nekbone* uses significantly more memory for VCRTs compared with the other miniapps (about 50% higher memory usage). The reason for this is the number of communicators created by this miniapp. Specifically, *Nekbone* creates seven duplicates of `MPI_COMM_WORLD`, which is more than other miniapps. Including the predefined and MPI internal communicators, that comes to a total of 24 communicators being used within *Nekbone*. While the duplicate communicators do not add too much memory usage for the user-visible communicators themselves, they add some additional memory usage for the MPI internal *node* and *node-roots* communicators. This results in a spurt in memory usage for VCRTs in *Nekbone*. This, of course, does not affect AV-Rankmap which retains the memory usage of the Rankmaps at only about 1.3 KB.

VI. Related Work

In [10], Balaji et al. discuss the memory overheads of communicators in MPI. They note that the memory usage of communicators increases significantly with the number of processes and affects the total number of communicators that can be created. They report that on IBM Blue Gene/P the number of new communicators that can be created on 128K processes drops to as low as 264 because of the memory usage associated with each communicator creation. These findings strengthen the argument for compressing the memory usage of network address management in communicators, that we address in this paper.

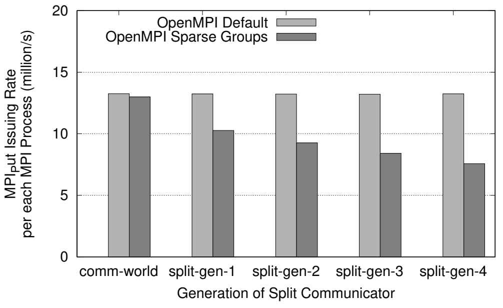


Fig. 19: Issue Rate for Different Generations of Communicators.

Several other works focus on reducing the memory usage of MPI communicators and groups [11], [20]. In order to support the various possible group patterns, these approaches use complex models for storing the ranks in groups. While these approaches provide similar reduction in memory usage as AV-Rankmap, they do so at the cost of high overhead in rank-address translation: this is a fundamental downside in these approaches. As mentioned earlier, in the performance-critical path, the overhead needs to be practically negligible for the proposed approach to be viable. As a specific example instantiation of these existing research papers, the sparse group proposed in [11] has been adopted as an optional feature in Open MPI. In this approach, the rank-address translation is from a child communicator to a parent communicator, rather than directly from a child communicator to the actual network address. This means that as more communicators are created, the library needs more time to iteratively traverse the tree of the ancestor communicators for network address lookup, thus significantly degrading performance. For a more quantitative comparison, we performed a case study on the performance of the sparse groups approach. In this comparison, we first performed an odd/even split on `MPI_COMM_WORLD` to create a first generation split communicator. Then we derived the next generation split communicator by splitting the current generation communicator in the same odd/even manner. Therefore, all split communicators have a stride mapping model. Figure 19 shows the theoretical maximum message issue rate on different generations of split communicators using the default Open MPI and Open MPI with sparse groups. Due to the iterative traversal of the tree of the ancestor communicators, Open MPI with the sparse group implementation has around 10% performance loss for each additional generation of communicators—with as few as four communicators, this adds up to nearly 40% performance overhead compared to the default Open MPI implementation. On the other hand, the performance of the default Open MPI is invariant to the generation of communicators.

Some studies proposed approaches to distribute the table for ranks among multiple processes [14], [15], [17]. Sack and Gropp [17] proposed a distributed algorithm for ordered child communicator construction that uses $O(n/p)$ memory by using distributed tables for storing the ranks. Recent work by Moody et al. [16] mentions a generalized `MPI_Comm_split`. They propose creating and storing process groups as chains in $O(1)$ memory and $O(\log n)$ construction time. They perform collectives by exchanging appropriate process ids during the operation. As we demonstrated in our case study, however, the lookup table is not necessary for most communicators. Also, because of the nature of the distributed lookup table, some rank-process translations need additional communication which adds a large performance overhead.

A more successful communicator memory compression technique used by MPICH, MVAPICH, Intel MPI, and many other MPI implementations was proposed by Goodell et al. [13]. This approach allows duplicate communicators to share the same VCRT as their parent, thus removing multiple copies of the same VCRT. As we mentioned earlier in the paper, while this approach eliminates the need for VCRTs in some limited cases, it is not applicable to most cases. Even for simple duplicate communicators, internal communicators used in MPI are not direct duplicates and thus cannot use this approach. Nevertheless, this approach is widely used in many MPI implementations and is, in fact, the baseline case that we compare against in this paper.

Compared with these previous studies, AV-Rankmap has four major advantages: (1) it eliminates the need for a lookup table for the majority of use cases; (2) it uses a simple process mapping model that avoids the overhead of complex mapping techniques and distributed mapping tables; (3) it tackles compression in both the network address and rank-address mapping structures; and (4) it performs such memory compression with no practically observable performance degradation.

VII. Concluding Remarks

In this paper we proposed a new mechanism, called AV-Rankmap, for network address management in MPI. AV-Rankmap detects patterns in rank-address mapping that applications naturally tend to have, as well as the fact that some parts of the network address structures are naturally more performance critical than others. It uses this information to compress the network address management structures. We demonstrated that AV-Rankmap significantly reduces the memory usage of communicators on large-scale systems with no practically observable degradation in performance.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

License

Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.

References

[1] Center for Exascale Simulation of Advanced Reactors. https://cesar.mcs.anl.gov.

[2] Center for Exascale Simulation of Combustion in Turbulence. https://cesar.mcs.anl.gov.

[3] CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks.

[4] Intel Software Development Emulator. https://software.intel.com/en-us/articles/intel-software-development-emulator.

[5] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[6] Nek5000. https://nek5000.mcs.anl.gov.

[7] OpenFabrics Interfaces (OFI). https://ofiwg.github.io/libfabric/.

[8] QBOX. http://computation.llnl.gov/projects/qbox-computing-structures-quantum-level.

[9] SNAP: SN Application Proxy. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/snap/.

[10] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on a Million Processors. *Parallel Processing Letters*, 21:45–60, 2011.

[11] Mohamad Chaarawi and Edgar Gabriel. Evaluating sparse data storage techniques for MPI groups and communicators. In *Proceedings of the 8th International Conference on Computational Science, Part I*, ICCS '08, pages 297–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, chapter Noncollective Communicator Creation in MPI, pages 282–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[13] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable Memory Use in MPI: A Case Study with MPICH2. In *Proceedings of the 18th EuroMPI Conference (EuroMPI)*, 2011.

[14] Humaira Kamal, Seyed M. Mirtaheri, and Alan Wagner. Scalability of Communicators and Groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[15] Humaira Kamal and Alan Wagner. An Integrated Fine-Grain Runtime System for MPI. *Computing*, 96(4):293–309, 2014.

[16] Adam Moody, Dong H. Ahn, and Bronis R. de Supinski. Exascale Algorithms for Generalized MPI_Comm_Split. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, 2011.

[17] Paul Sack and William Gropp. A Scalable MPI_Comm_Split Algorithm for Exascale Computing. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, 2010.

[18] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

[19] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 665–676, 2015.

[20] Jesper Larsson Träff. Compact and Efficient Implementation of the MPI Group Operations. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, pages 170–178, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.